Design of On-Chip Cores and Sensors to Improve Embedded System Reliability

Fabian Vargas obtained his Ph.D. Degree in Microelectronics from the Institut National Polytechnique de Grenoble (INPG), France, in 1995. In 2005, he spent a sabbatical year at the Technical University of Lisbon (INESC-ID / IST), Portugal. At present, he is Associate Professor at the Catholic University (PUCRS) in Porto Alegre, Brazil.

His main research domains involve the HW-SW co-design and test of system-on-chip (SoC) for critical applications; system-level design and methodologies for radiation and electromagnetic compatibility; and the on-chip sensor design for reliability and aging binning.

Among several activities, Prof. Vargas has served as Technical Committee Member or Guest-Editor in many IEEE-sponsored conferences and journals. He holds 6 BR and international patents, co-authored a book and published over 200 refereed papers. Prof. Vargas is associate researcher of the BR National Science Foundation since 1996.

He co-founded the IEEE Latin American Test Technology Technical Council (LA-TTTC) in 1997 and the IEEE Latin American Test Workshop (LATW) in 2000. Prof. Vargas received the Meritorious Service Award of the IEEE Computer Society for providing significant services for chairing the IEEE Latin American Regional TTTC Group and the LATW for several years. Prof. Vargas is a Golden Core Member of the IEEE Computer Society.
Design of On-Chip Cores and Sensors to Improve Embedded System Reliability

Fabian Vargas

vargas@pucrs.br
evargas@computer.org
Development of IP Cores

Development of On-Chip Sensors
Development of IP Cores

Development of On-Chip Sensors
Nowadays, *embedded systems* must respect *strict timing constraints* to support real-time (RT) applications.

![Image](image-url)

They have to provide *logically* and *temporally correct* results!

In this reality, these systems mandate the adoption of Real-Time Operating Systems (RTOS) that manage *task switching process, concurrency between tasks, memory, time* as well as *interrupts.*

vargas@computer.org
Understanding the Problem ...

The **increasing hostility of the electromagnetic environment** caused by the widespread adoption of electronics, (mainly **wireless technologies**), represents a huge challenge for the reliability of RT embedded systems.

- **Radiated electromagnetic interference (EMI)**
- **Power Supply Disturbances (PSD)**
- **Transient Faults**

These faults can affect not only the **applications** running on embedded systems but also the **RTOS** executing the application code, by causing **scheduling dysfunctions** that could lead to incorrect system behavior.
Several solutions have been proposed. However, they provide fault tolerance only at the application level and do NOT consider faults affecting the RTOS that propagate to application tasks.

e.g.: about 34% of the faults injected in processor’s registers led to scheduling dysfunctions:

- 44% of these dysfunctions led to system crashes,
- 34% caused RT problems and
- 22% generated incorrect outputs (propagate to system outputs).

If not detected at the RTOS-level, these faults escape detection by conventional (app-level) techniques as well!
In this context…

We have been developing an IP core (RTOS-Guardian: RTOS-G) to monitor RTOS activity in embedded systems.

The RTOS-G targets faults that ESCAPE detection by the native structures present in the RTOS kernel.
1. Preliminaries

RTOS Preemptive Algorithm’s Behavior

Task1 and Task2 have the same priority and they do not possess external dependencies.

While a third task (Task3) with higher priority is blocked and waiting for an **external event**. But during Task1 execution this dependency is solved.

Then, the RTOS stops the current task (Task1) execution and switches to Task3.

Since this task has the highest priority, it will be executed completely before returning to the interrupted Task1.
1. Preliminaries

From the Preemption Algorithm to the Specification of the RTOS-G

✓ The RTOS-G must keep a complete list of all tasks labeled ready (ready-list) and blocked (blocked-list).

✓ Every time a scheduling event is performed, the ready task marked with the highest priority must be executed.

✓ The complexity of monitoring this kind of behavior relies on keeping track of the ready-list: its elements must NOT have any pending IO requests or semaphore objects still to be acquired.

✓ Then, the RTOS-G monitors not only the task addresses, but also the addresses related to the kernel synchronization, including: SemaphoreLock() and SemaphoreUnlock().
1. Preliminaries

and the following execution flow analysis is adopted …

The RTOS-G observes the order in which the functions responsible for task synchronization are being called by the RTOS to infer ready-list constraints.

- Task1 is running and tries to acquire a semaphore. Then, system call is performed.
- The RTOS kernel realizes that the semaphore is already locked. In order to prevent the system from going into a deadlock as well as to increase the CPU usage, the kernel performs a CS calling another task into execution.
- When the RTOS-G detects this flow, it will infer that Task2 is the next to run and therefore it should be taken out from the ready-list.
- In conclusion: the execution flow for an already locked semaphore consists of: SemaphoreLock() and ReSchedule().
- A similar analysis can be performed for other situations, always concentrating all efforts in keeping the detection algorithm generic enough for any RTOS or processor.
2. The Proposed Approach

Block diagram of the target embedded system

Events: Tick, interruption, ...
(Reference for Task Context Switching)

RTOS-G identifies the current task under execution by correlating addresses flowing through the bus with the information stored in an Address Table generated during the compilation process.

Memory Addresses accessed by the processor.
2. The Proposed Approach

Block diagram of the RTOS-G

TC identifies the task in execution based on the address accessed by the CPU during code execution.

CAM1: ready-list
CAM2: blocked-list.

FI analyses the scheduling process execution order and identifies the event that triggered the process (e.g., occurrence of a Tick signal, IO request or semaphore acquisition/release).

LMEG classifies all tasks in two separate lists, ready tasks and blocked tasks, each one organized according to their priority and indicates errors.

vargas@computer.org
2. The Proposed Approach

TIME constraint is handled by the core …

by redefining the Context Switching Parameter and creating a new Time Limit Variable:

Time for Context Switching (\(\Delta \text{time, proportional to the number and complexity of resources, services, used by the RTOS}\))

![Diagram of context switching](image-url)
2. The Proposed Approach

Regarding the fault detection capability, the RTOS-G targets **two types of faults**:

**Sequence error (E_seq):** occurs at the **end of the Time Limit, \( tl \)**, by noting that the **current task is not the expected one** according to the task’s execution flow (\( ready-list \) and \( priority-list \)).

**Time error (E_time):** occurs when a **task switching process takes place in between two consecutive context switching events** (e.g., two consecutive \( ticks \)) thus, violating the time constraints associated to the RT-system.
2. The Proposed Approach

Some features …

RTOS-G area overhead (333 LUTs) with respect to the Plasma (MIPS-like) microprocessor (3306 LUTs): 10%.

For larger cores, area overhead can be reduced to values lower than 10% because the size of the I-IP core is basically dependent on the kernel complexity (i.e., number of resources, or functions, existing in the RTOS kernel and how they are interconnected).

CAMs are able to store information up to 14 tasks simultaneously in execution (both account for 60% of the I-IP overhead).

CAMs can be placed externally: RTOS-G’s overhead is 4%.

RTOS-G fault latency: 2% of the latency yielded by the RTOS (measurement with ChipScope in terms of processor CC).
3. Practical Experiments

**Case Study:**

Von Neumann 32-bit RISC Plasma microprocessor running a RTOS ([opencores.org](http://opencores.org)).

- Plasma’s instruction set compatible to MIPS architecture.
- 3 test programs were developed to exploit different services offered by the Plasma’s RTOS:

  **Increasing use of RTOS services**

  8 tasks access and update the value of a variable, which is protected by a semaphore. Other variable is accessed by an int. The 8 tasks priorities: "1, 2, 3, 4, 1, 2, 3, 4". The int has the maximum priority.

  4 tasks access and update the value of a variable, which is protected by a semaphore. T5 communicates with T6 through a message queue. An int accesses a variable. The 6 tasks priorities: "1, 2, 3, 4, 5, 6". The int has the maximum priority.

  2 tasks access and update the value of a global variable, which is protected by a semaphore. T3 communicates with T4 through a message queue. T5 and T6 access and update the value of a variable, which is protected by a mutual exclusion semaphore (MUTEX). An int communicates with a T7 throughout a message queue. The 7 tasks priorities: "1, 2, 3, 4, 5, 6, 7". The int has the maximum priority.
3. Practical Experiments

Test board designed for IEC 62.132-2 and 61.004-29 electromagnetic susceptibility analysis
3. Practical Experiments

Fault injection campaigns: *generated according to the IEC 61.000-4-29 Int. Std. for conducted EMI on the DC input power port of FPGA 1*

Voltage dips were randomly injected at the FPGA 1 $V_{dd}$ input pins at a frequency of 25.68 kHz and consisted of dips of about 10.83% of the nominal $V_{dd}$.

Injected noise at the FPGA power bus (conducted EMI)
3. Practical Experiments

TABLE I
Results for Fault Detection

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>RTOS Kernel [%]</th>
<th>RTOS-G [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>BM1</td>
<td>2.40</td>
<td>99.90</td>
</tr>
<tr>
<td>BM2</td>
<td>25.90</td>
<td>100.00</td>
</tr>
<tr>
<td>BM3</td>
<td>45.80</td>
<td>100.00</td>
</tr>
<tr>
<td>Average</td>
<td>24.70</td>
<td>99.97</td>
</tr>
</tbody>
</table>

As long as more complex services of the kernel are used, the higher RTOS error detection.

The native fault detection mechanism (assert() function) is called by the kernel every time RTOS runs its services (message queues, semaphores).

The priority level of tasks using the same RTOS services has no direct influence on the error probability.
3. Practical Experiments

Fault injection campaigns: radiation on a Gamma Cell with Co$^{60}$

<table>
<thead>
<tr>
<th>Experiment</th>
<th>FPGA0 (krads)</th>
<th>FPGA1 (krads)</th>
<th>FPGA2 (krads)</th>
<th>FPGA3 (krads)</th>
<th>FPGA4 (krads)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st experiment, December/2010</td>
<td>0</td>
<td>5.6</td>
<td>51.9</td>
<td>111.0</td>
<td>216.0</td>
</tr>
<tr>
<td>2nd experiment, March/2011</td>
<td>0</td>
<td>217.6</td>
<td>263.9</td>
<td>323.0</td>
<td>---</td>
</tr>
</tbody>
</table>
3. Practical Experiments

FPGA1

FAULTS DETECTED

RTOS: 18,00%

RTOS-G: 68,67%

FPGA2

FAULTS DETECTED

FPGA3

FAULTS DETECTED
A second work ...
Case Study & Practical Experiments

SoC General architecture

Microprocessor

Xilinx FPGA

Operating System (OS)

Driver

Error Indication

uCOS-II

IP Core

SoC General architecture
Case Study & Practical Experiments

IEC 62.132-2 std compliant board.
Four-layers: Gnd (top) - signal - signal - Vdd (bottom).
Case Study & Practical Experiments
Case Study & Practical Experiments

Test setup:
Frequency range: 80MHz – 3GHz
EM field: 10V/m – 220V/m

Comparison among techniques

Detected mostly faults that affected data-flow execution and configuration logic of the FPGA, as well as semaphore-related ones.

Detected mostly faults that affected data-flow execution.

Detected only faults that prevented the OS from controlling the semaphores between tasks.

Errors detected by the IP Core
Errors detected by different techniques (check-point insertion)
Implemented in the application code
Detected mostly faults that affected data-flow execution and configuration logic of the FPGA, as well as semaphore-related ones.

IEC 62132-2 Experimental Results
(Prime Numbers Generator)
An initial work ...
Initial Work

General Block Diagram of the I-IP (RTOS-G) under development for MPSoCs
An initial study ...
Despite of more than 20 years of highly active research on WCET analysis, there are still serious open problems in the area. The most challenging problem is the high complexity of today’s processors: features like cache coherence (hit and misses) and pipeline (expeculative execution) create a huge state space.

Static WCET Analysis tools & Measurement-Based WCET Analysis tools have evolved as complementary approaches, but both suffer from the state problem (i.e., the precise determination of possible processor states at different program locations).
We are studying the development of an IP core* dedicated to:

(a) perform on-chip WCET measurement as a complement to Static WCET Analysis;
(b) identify timing-critical parts of the SW (e.g., functions, instructions) and the timing-critical parts of the HW (e.g., cache coherence, pipeline execution) for further design optimization purposes;
(c) on-chip, runtime WCET monitoring.

Initial Study

Tasks for determining the WCET of a program (Static Analysis):

- Executable Program
- CFG Building
- Value analysis
- Loop bound analysis
- Cache analysis
- Pipeline analysis
- WCET Bound
- Paph analysis
Initial Study

**CFG Building** decodes, i.e. identifies instructions, and reconstructs the control-flow graph (CFG) from a binary program.

**Value Analysis** computes value ranges for registers and memory cells, and address ranges for instructions accessing memory. Value analysis tries to determine the values in the processor memory for every program point and execution context. Its results are used to determine possible addresses of indirect memory accesses —important for cache analysis— and in loop bound analysis.

**Loop Bound Analysis** determines upper bounds for the number of iterations of simple loops. WCET analysis requires that upper bounds for the iteration numbers of all loops be known.

**Cache Analysis** classifies memory references as cache misses or hits.

**Pipeline Analysis** predicts the behavior of the program on the processor pipeline. Pipeline analysis models the pipeline behavior to determine execution times for sequential flows (basic blocks) of instructions.

**Path Analysis** determines a worst-case execution path of the program. Using the results of the micro-architecture analyses, path analysis determines a safe estimate of the WCET. The program’s control flow is modeled by an integer linear program [14] so that the solution to the objective function is the predicted worst-case execution time for the input program.

The RTOS-G checks if the **execution time of a given task** fits inside the **estimated WCET**.

**Task1** and **Task2** have the same priority and that they do not possess external dependencies. 

While a third task (**Task3**) with higher priority is blocked and waiting for an external event, but during **Task1** execution this dependency is solved.

Since this task has the highest priority, it will be executed completely before returning to the interrupted **Task1**.
The RTOS-G checks if the **time required by the RTOS to reschedule due to a semaphore already taken** fits inside the estimated WCET.

Task1 is running and tries to **acquire a semaphore**. Then, system call is performed.

The RTOS kernel realizes that the semaphore is already locked. In order to prevent the system from going into a deadlock as well as to increase the CPU usage, the kernel performs a **CS** calling another task into execution.

When the RTOS-G detects this flow, it will infer that **Task2 is the next to run** and therefore it is taken out from the ready-list.

In conclusion: the execution flow for an **already locked semaphore** consists of: **SemaphoreLock()** and **ReSchedule()**.

A similar analysis can be performed for other situations, always concentrating all efforts in keeping the detection algorithm **generic enough** for any RTOS or processor.